Integration_types

Термин интеграция имеет широкое значение. Под ним можно понимать объединение информационных систем, приложений, различных компаний или людей.

В зависимости от поставленных задач интеграция может осуществляться на четырёх **уровнях**: на уровне платформ, данных, приложений и бизнес-процессов.

Интеграция платформ

Этот тип интеграции обеспечивает взаимодействие между приложениями на разных программных платформах, а также работу этих приложений на сторонней платформе. При интеграции платформ используются такие технологии, как виртуализация, ПО промежуточного слоя и удалённый вызов процедур.

Интеграция данных

Интеграция на уровне данных предполагает совместное использование данных различных систем. Интеграция данных может оказаться проще, чем интеграция приложений, так как СУБД имеют развитые возможности программного доступа к данным из других приложений. Наиболее распространённые технологии этого класса — ODBC, JDBC, ADO.NET. Кроме того, на сегодняшний день широко распространены технологии ORM, которые позволяют абстрагироваться от деталей взаимодействия с конкретными СУБД.

Интеграция приложений

Интеграция на уровне приложений подразумевает использование готовых функций приложений в других системах. При таком типе интеграции чаще всего используются следующие технологии: интерфейсы прикладного программирования, обмен сообщениями (корпоративная сервисная шина), сервис-ориентированная архитектура (SOA) и интеграция пользовательских интерфейсов.

Интеграция бизнес-процессов

Наиболее целостным подходом к интеграции систем является интеграция бизнес-процессов. В рамках интеграции бизнес-процессов происходит

интеграция и приложений, и данных, и, что не менее важно, людей, вовлечённых в этот бизнес-процесс.

Так же можно выделить следующие разделения видов интеграций:

1. **Внутренняя (Enterprise Application Integration) и внешняя (Business-to-Business Application).**

   Интеграция подразделяется на внешнюю и внутреннюю. Внутренняя интеграция подразумевает объединение корпоративных приложений в одной организации (Enterprise Application Integration), а внешняя – интеграцию информационных систем организаций (Business-to-Business Application).

2. **Простая и сложная** (очень условное название)

   Интеграция может быть с использованием no-code инструментов (например, Integromat), а может требовать разработчиков и написания кода. Вот тут классная подборка no-code инструментов.

   Кстати, можно выделить еще ручную интеграцию. Например, сотрудник склада, переносящий данные из системы 1С:Склад в CRM, занимается интеграцией этих систем.

3. **Классификация по целям**

   Интеграция может иметь разные цели. В основном, речь идёт про организацию сквозного бизнес-процесса (процесс заказа продуктов, например, требует работы нескольких систем). Или про организацию хранения данных (MDM-класс систем, DWH-системы и т.п.). Или про представление данных в одном окне : всё это ради экономии времени, денег или предоставлении какой-то новой, ранее невозможной услуги (сервиса) или продукта

4. **Интеграция может быть стихийной или плановой**

   Вам срочно надо объединить две системы, чтобы организовать процесс? Фигачим! О, неплохо бы ещё систему CRM к ним интегрировать? Делаем! И так год за годом. Куча систем, как клубок, наматывается: то там интегрируем, то тут. Плановая, соответственно, возникает, когда есть план)

Плановая интеграция

Интеграция Ad-hock

5. **Синхронная или асинхронная интеграция**

   Асинхронная интеграция нужна — это как сообщение в почте или чате, ответа мы не ждём сразу. Синхронная — это как общение по телефону: если ты звонишь человеку, то ты общаешься с ним и ждёшь ответа сразу во время разговора.

6. **Интеграция систем, подсистем, сервисов, микросервисов**

   Если вы интегрируете свою систему CRM с чужой системой MDM, то это интеграция систем.

   Если сервису Яндекс.Кошелек вдруг нужны данные с сервиса Госуслуги, то перед нами интеграция сервисов.

   Если наша система построена по SOA, то мы имеем распределенную систему, подсистемы которой интегрируются между собой (например, с помощью шины). Для систем на микросервисах все тоже самое.

7. **Паттерны интеграции по типу обмена данными**

   Речь идет о таких паттернах, как обмен файлами, обмен через общую базу данных, удалённый вызов процедур, обмен сообщениями. Вот в этой книге можно почитать про все эти паттерны (глава 2).

   Самая простая (и исторически возникшая самой первой) — это интеграция по паттерну **«общая база данных»**. Быстро, дешево, небезопасно. Не используйте этот тип интеграции, если у вас есть хоть какая-то альтернатива.

«Обмен файлами» — это простой способ передать файлы. Выкладываете отчет на FTP-сервер как на гугл-диск, а потом другая система его скачивает. Дешево, достаточно безопасно, но сложного взаимодействия не построить (попробуйте впятером работать с одной страницей в гугл-док, не общаясь другими каналами связи.

На основе **«удалённый вызов процедур (RPC)»** можно построить куда больше взаимодействий. Сейчас это самая популярная тема и по ней пару предложений написать недостаточно. Поэтому просто скажу, что из всего многообразия сейчас максимально популярны gRPC, REST, SOAP, GraphQL (формально не все из этого RPC). Если вам нужно построить интеграцию между двумя системами, то скорее всего это REST — вот и все, что стоит пока запомнить.

Паттерн **«обмен сообщениями»** становится необходим в одном из таких случаев, например

✓ крупный проект, в котором мы хотим иметь единую точку для управления интеграциями

✓ асинхронном взаимодействии

✓ больших потоках информации

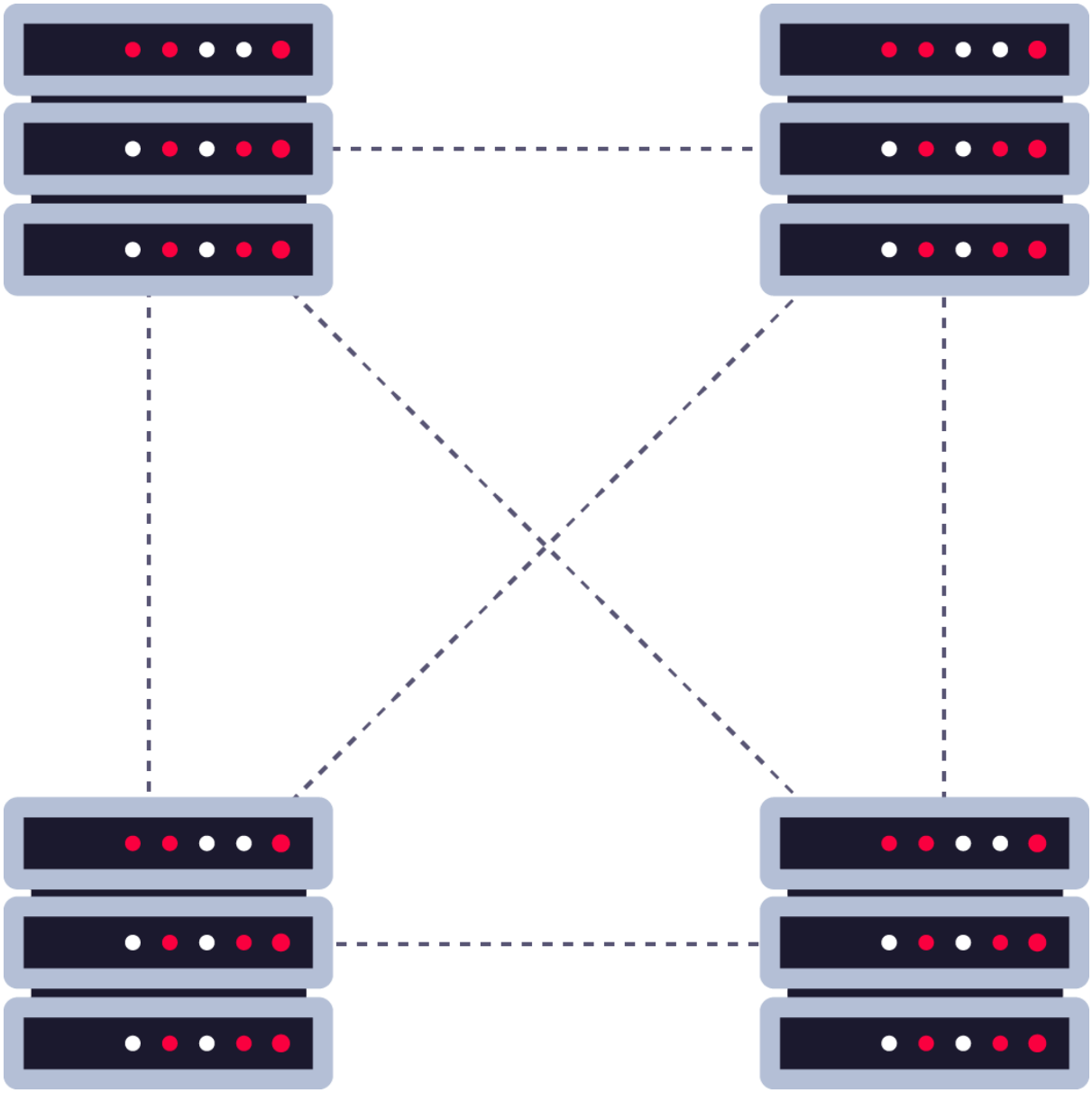
✓ масштабировании

Вот тогда появляется сервисная шина предприятия или брокер. Это дорого, долго, но порой необходимо.

Интеграции могут отличаться по структуре связи:

- Точка-точка (Point-to-Point).
- Шлюз (hub-and-spoke).
- Шина (Bus).

Интеграция по типу «точка точка»



Интеграция приложений напрямую, является методом интеграции, при котором взаимодействие между системами происходит без применения универсального централизованного посредника, такого, как сервисная шина предприятия (ESB).

Плюсы:

1. Сравнительная низкая трудоемкость реализации.
2. Возможность повышения производительности взаимодействия, достигаемая при

отсутствии посредников и возможности применения низкоуровневых протоколов.

Недостатки:

- Увеличение трудоемкости разработки и поддержки интеграционных решений по мере роста количества интегрируемых систем.
- Отсутствие гибкости интеграционного решения из-за жестких связей между системами.
- Отсутствие унифицированного инфраструктурного обеспечения интеграционного взаимодействия (мониторинг, логирование, безопасность и т. д.).

Децентрализованное соединение точка-точка означает, что интегрируемые приложения устанавливают прямые связи друг с другом. Данная модель обычно используется на начальной стадии любого интеграционного проекта как наиболее простой подход. Как видно из рис. 1,а, отличительной особенностью рассматриваемой модели являются многочисленны связи (интерфейсы) между приложениями, которые повышают трудоемкость управления корпоративной информационной системой. Это связано с тем, что с вводом нового приложения количество новых интерфейсов растет по формуле n (n–1), где n — число приложений. При модификации одного из приложений корпоративной информационной системы возможно усложнение поддержки всей системы, в результате может снизиться эффективность принятия решений и управления бизнес-процессами.

Задача интеграции «точкаточка» относительно проста. Нужно понять, каким образом каждая из двух взаимодействующих систем готова передавать и получать данные, создать соответствующие технические решения для обращения к этим интерфейсам, а также реализовать механизм преобразования данных из формата системыисточника в формат системы-приемника. В лучшем случае информационные системы предоставляют для интеграции специальный программный интерфейс (API), а в худшем - чтение и запись информации приходится производить непосредственно в базу данных приложения. В результате возникает локальное интеграционное решение - некий обособленный программный модуль собственной разработки со всеми вытекающими требованиями к его обслуживанию и поддержанию актуальности.

Такое интеграционное взаимодействие не составляет большую проблему до тех пор, пока интеграций «точкаточка» мало - одна-две. Однако практика показывает, что количество интеграций «точкаточка» имеет склонность возрастать, а качество управления этими интеграциями - наоборот, стремительно падать. Причин тому много: возрастает число модулей

интеграции, из организации уходят разработчики, делавшие тот или иной модуль, изменяются форматы данных в интегрируемых системах и т.д. Печальным итогом эволюционного развития интеграций «точкаточка» является сложнейший «фарш» интеграционных взаимодействий между приложениями предприятия, отношение к которому сотрудников ИТ-подразделения проще всего выразить в нескольких словах: «Пока работает - лучше не трогать». Однако такая ситуация не устраивает ни само ИТ-подразделение, ни бизнес-заказчиков.

Использование шлюза Hub-and-Spoke

Интеграционный шлюз - это маршрутизатор сообщений, связи которого можно поделить на "внешние" и "внутренние".

Эти связи могут работать по одному и тому же протоколу - в таком случае смысл шлюза в разбиении общей шины на сегменты с целью добавления отказоустойчивости или защищенности. К примеру, при наличии филиалов в разных городах, в каждый можно "поставить" шлюз, который будет передавать сообщения в головной офис по шифрованному каналу.

В таком случае общая интеграционная шина окажется разбита на сегменты, которые могут продолжать ограниченно функционировать при падении интернет-канала.

Кроме того, интеграционный шлюз может соединять приложения, работающие по разным протоколам - в таком случае он будет еще и адаптером.

Hub-and-Spoke - наиболее эффективная топология для репликаций, потому что она минимизирует движение данных в сети, особенно в больших организациях. Hub-and-Spoke устанавливает один центральный сервер Hub, который намечает и начинает все репликации со всеми другим серверами типа Spokes. Spokes сервера передают на Hub сервер все свои изменения и почту, а Hub в свою очередь передает все изменения на каждый другой сервер. Hub сервера реплицируют изменения между собой или на мастер Hub серверов, в организациях которые используют больше чем один Hub. Короче говоря, Hub сервер действует как менеджер репликаций и почты в системе.

Чтобы наметить репликации в системе по типу Hub-and-Spoke, Вы создаете один документ подключения для каждой связи Hub-and-Spoke. В каждом документе подключения, Hub сервер - всегда сервер источник, а сервер назначения, всегда один из серверов Spoke.

Топология Hub-and-Spoke может быть особенно полезна, в целом для нескольких серверов, в централизованном офисе, который должен соединиться через телефонные или выделенные линии с региональными офисами. Если Вы имеете большой участок сети, Вы можете использовать комбинацию топологий Hub-and-Spoke и Peer-to-Peer между двумя Hub серверами.

Применение топологии репликаций Hub-and-Spoke:

Установите несколько протоколов на Hub сервера, чтобы позволить связь в Domino системе, которая будет использовать больше чем один протокол. Hub сервер может соединять с несколькими Notes поименованными сетями, где имеются отдельные Hub сервера.

Используйте несколько частей сети - например, LAN и WAN.

Централизуйте администрирование Domino Directory, ACL стандартизируйте базы данных. Ограничьте доступ на Hub сервера. Вы можете определить в ACL, Hub серверам - с доступом Менеджера, а Spokes - доступ Читателя, чтобы делать изменения в одной точной копии на Hub сервере.

Разместите программы серверов, например агентов, на Hub серверах, чтобы делать их легко доступными.

Соедините удаленные участки сети с Hub серверами.

Минимизируйте движение сети, и максимизировать эффективность сети.

Централизуйте данные, создайте резервные Hub сервера.

Улучшите балансировку нагрузки на серверов. Однако движение по сети увеличивает нагрузку на сегменты сети с установленными серверами Hub. Если Вы имеете больше чем 25 серверов, установите несколько Hub серверов. Если Hub сервер выключается, репликации для этих Hub и Spokes будут недоступны, пока Hub сервер не будет восстановлен или заменен.

Примечание. Не использует Hub-and-Spoke репликации для баз данных, размер которых большей чем 100МБ. Реплицируйте базу данных непосредственно между серверами, намечая репликации для этой базы данных в документе подключения.

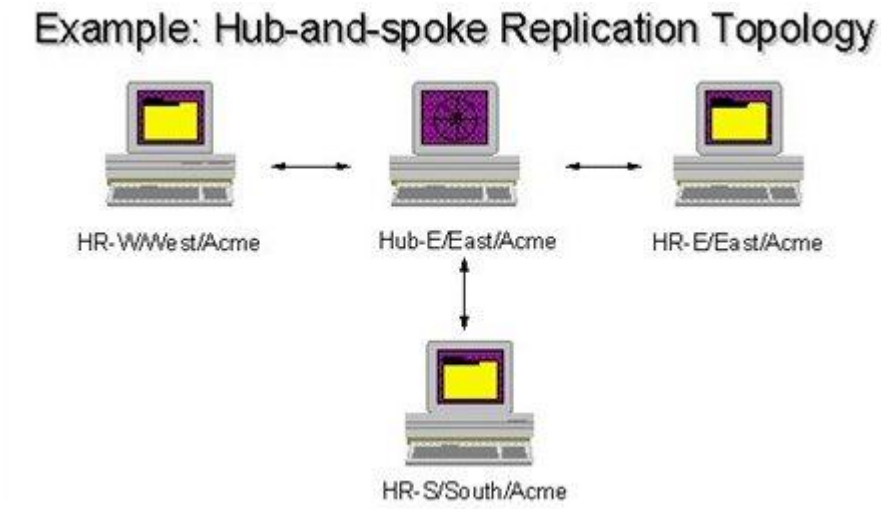


Рис. 33 Пример использования топологии Hub-and-Spoke для репликаций.

В этом примере, Acme корпорация имеет один Hub сервер, Hub-E/East/Acme и три Spoke сервера. Spoke сервера - HR-E/East/Acme, HR-S/South/Acme, HR-W/West/Acme - содержат служебные приложения. Любые изменения, замены в приложениях реплицируются через Hub-E/East/Acme на HR сервера. HR сервера посылают изменения, замены Hub, которые в свою очередь посылает изменения на все HR сервера, используя для этого Hub.

С тремя документами подключения, которые Acme создал, Hub сервер исполняет репликации, сокращая нагрузку на Spokes сервера. При создании приложений, доступных на East, West, пользователи South запрещают их распространение через дорогостоящие связи WAN.

Шина

Интеграционная шина - это прежде всего среда передачи сообщений, позволяющая доставлять сообщения на основе логических признаков. "На пальцах", это означает, что в самих сообщениях не содержатся никакие физические адреса.

Простейшие варианты интеграционных шин держат у себя справочник соответствия логических и физических адресов. Логические адреса формируются исходя из назначения приложения, его названия или оргструктуры. Этого достаточно чтобы называться шиной :)

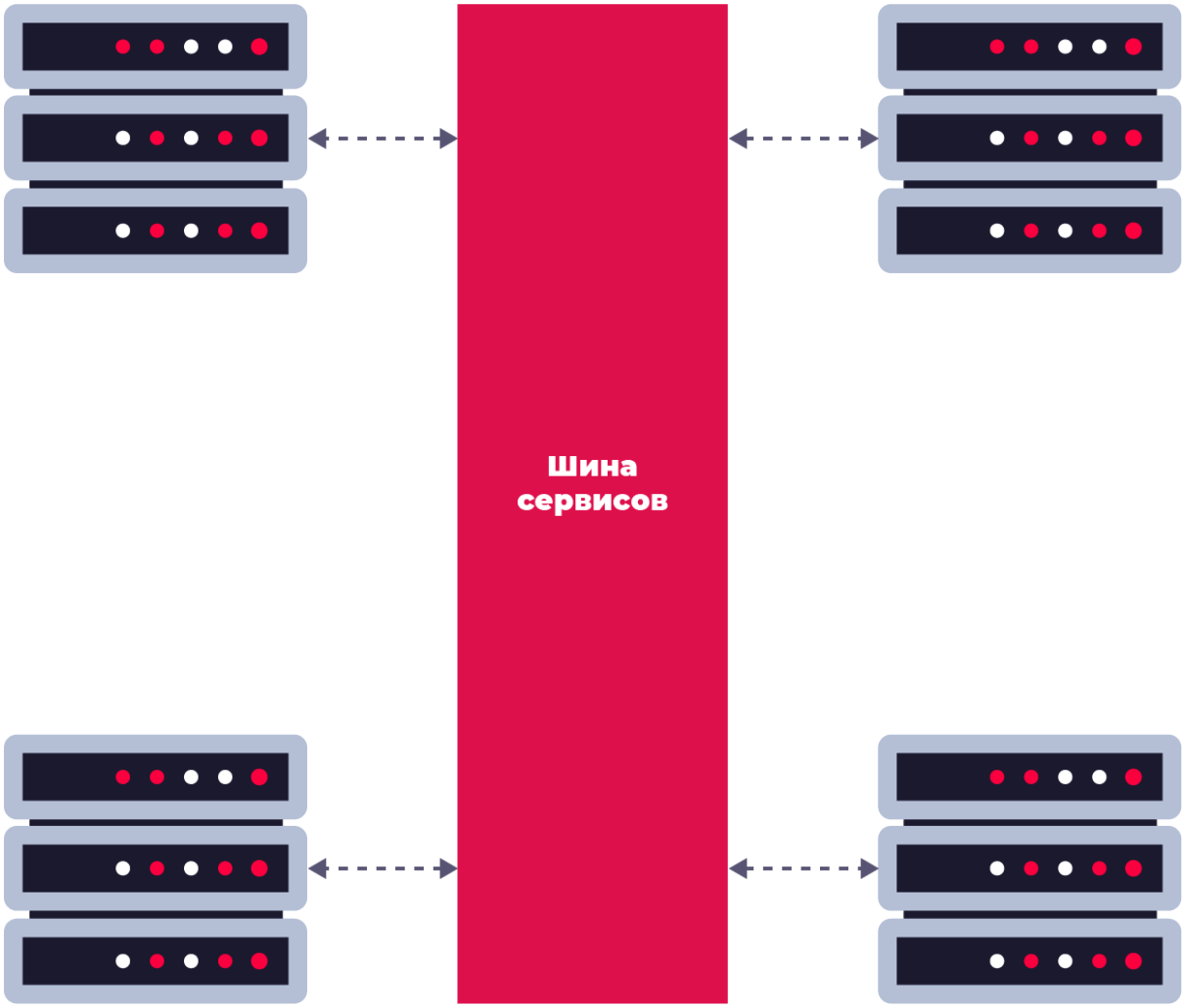
Более сложные варианты дают возможность доставки сообщений по системе pub/sub (публикатор - подписчик).

Интеграционная шина может быть построена вокруг центрального узла (или нескольких узлов), где "крутится" маршрутизатор (см. далее) - или же быть распределенной, существуя в виде кучи адаптеров (см. далее) на стороне каждого приложения с общей базой адресов.

Единая сервисная шина

Связующее ПО, обеспечивающее обмен данными между различными информационными системами компании. У такого решения есть масса преимуществ, таких как гибкость, возможность масштабирования и централизация контроля. При этом важным требованием к подобного рода ПО является возможность интеграции с системами других организаций — поставщиками, клиентами, партнёрами.

Интеграция посредством единой сервисной шины



Так же по теме

Маршрутизатор сообщений - это приложение, смысл существования которого - передавать сообщения от одного приложения к другому.

Обычно маршрутизатор сообщений можно найти внутри интеграционный шины - но не каждый маршрутизатор образует вокруг себя шину. Как минимум, нужна работа с логическими адресами.

Также маршрутизаторы сообщений иногда реализуют другие инфраструктурные сервисы, вроде гарантированной доставки, ведения журнала сообщений или преобразования форматов.

Интеграционный шлюз - это маршрутизатор сообщений, связи которого можно поделить на "внешние" и "внутренние".

Эти связи могут работать по одному и тому же протоколу - в таком случае смысл шлюза в разбиении общей шины на сегменты с целью добавления отказоустойчивости или защищенности. К примеру, при наличии филиалов в разных городах, в каждый можно "поставить" шлюз, который будет передавать сообщения в головной офис по шифрованному каналу.

В таком случае общая интеграционная шина окажется разбита на сегменты, которые могут продолжать ограниченно функционировать при падении интернет-канала.

Кроме того, интеграционный шлюз может соединять приложения, работающие по разным протоколам - в таком случае он будет еще и адаптером.

Адаптер - это маршрутизатор сообщений, разные связи которого работают по разным протоколам.

Брокер - это маршрутизатор сообщений повышенной сложности. К примеру, он может работать с децентрализованной шиной. Или выполнять хитрые трансформации, выступая универсальным центральным адаптером.

Он соединяет в себе средства разработки, масштабируемую среду исполнения и средства моделирования. Взаимодействие между компонентами брокера часто базируется на очередях. Системы очередей сообщений предоставляют асинхронный метод взаимодействия для прикладных программ. Приложения обмениваются данными, посылая и принимая их в виде сообщений. При этом приложения не взаимодействуют друг с другом напрямую, а обращаются к службе очередей сообщений.

Как можно заметить, ни один терминов не исключает другие. Вполне можно себе представить счастье архитектора-перфекциониста в виде мега-интеграционной шины, построенной на маршрутизаторе сообщений, который является брокером, шлюзом и адаптером. Он будет уметь работать по любому протоколу, делать SQL-запросы, проводить XSLT-трансформации - и станет кошмарным сном любого разработчика :)

А отсюда есть еще один вывод: любую достаточно сложную программу можно обозвать любым словом. Поэтому, чтобы коллеги вас понимали - старайтесь называть ее теми словами, которые используются в официальном названии программы.

К примеру, WebSphere Message Broker имеет в своем названии слово "Брокер" (и заслуженно носит его за свою непомерную крутизну) - поэтому и называть его надо брокером, а не маршрутизатором сообщений, адаптером, шлюзом или шиной.

Интеграционное тестирование: что это? Виды, примеры. Что такое интеграционное тестирование? Зачем оно нужно? Примеры, подходы, стратегия и методологии...

источник https://logrocon.ru/news/intgration_testing

Что такое интеграционное тестирование?

Интеграционное тестирование

– это тип тестирования, при котором программные модули объединяются логически и тестируются как группа. Как правило, программный продукт состоит из нескольких программных модулей, написанных разными программистами. Целью нашего тестирования является выявление багов при взаимодействии между этими программными модулями и в первую очередь направлен на проверку обмена данными между этими самими модулями. Именно поэтому оно также называется «I & T» (интеграция и тестирование), **«тестирование строк»** и иногда «тестирование потоков».

Зачем нужно интеграционное тестирование?

Каждый программный модуль проходит отдельные этапы тестирования (модульное тестирование), но не смотря на это, дефекты могут оставаться по ряду причин:

- Поскольку, как правило, модули разрабатываются разными специалистами, их понимание и логика программирования могут отличаться. Тут интеграционное тестирование становится необходимым для проверки взаимодействия модулей между собой.
- Во время разработки модуля заказчики часто меняют требования, и если у вас сжатые сроки требования могут попросту не успеть пройти модульное тестирование, и, следовательно, системная интеграция может пройти

с помехами. Опять получается, что от интеграционного тестирования не убежать.

- Интерфейсы программных модулей с базой данных могут быть ошибочными
- Внешние аппаратные интерфейсы, если таковые имеются, могут быть ошибочными
- Неправильная обработка исключений может вызвать проблемы.

Примеры интеграционного тестирования

Интеграционное тестирование отличается от других видов тестирования тем, что он сосредоточен в основном на интерфейсах и потоке данных (между модулями). Здесь приоритет проверки присваивается интегрирующим ссылкам, а не функциям блока, которые уже проверены.

Пример тестирования интеграции для следующего сценария:

Приложение имеет 3 модуля, например «Страница входа», «Почтовый ящик» и «Удалить электронную почту». Каждый из них интегрирован логически.

Здесь нет нужды тестировать страницу входа, т.к. это уже было сделано в модульном тестировании. Но проверьте, как это интегрировано со страницей почтового ящика.

Аналогично, «Почтовый ящик»: проверьте его интеграцию с модулем «Удалить электронную почту».

Стратегии, методологии и подходы в интеграционном тестировании

Программная инженерия задает различные стратегии интеграционного тестирования:

- Подход Большого взрыва.
- Инкрементальный подход:
  - Нисходящий подход (сверху вниз)
  - Подход «снизу вверх»
  - Сэндвич – комбинация «сверху вниз» и «снизу вверх»

Ниже приведены различные стратегии, способы их выполнения и их ограничения, а также преимущества.

Подход Большого взрыва

Здесь все компоненты собираются вместе, а затем тестируются.

Преимущества:

- Удобно для небольших систем.

Недостатки:

- Сложно локализовать баги.
- Учитывая огромное количество интерфейсов, некоторые из них при тестировании можно запросто пропустить.
- Недостаток времени для группы тестирования, т.к тестирование интеграции может начаться только после того, как все модули спроектированы.
- Поскольку все модули тестируются одновременно, критические модули высокого риска не изолируются и тестируются в приоритетном порядке. Периферийные модули, которые имеют дело с пользовательскими интерфейсами, также не изолированы и не проверены на приоритет.

Инкрементальный подход

В данном подходе тестирование выполняется путем объединения двух или более логически связанных модулей. Затем добавляются другие связанные модули и проверяются на правильность функционирования. Процесс продолжается до тех пор, пока все модули не будут соединены и успешно протестированы.

Поэтапный подход, в свою очередь, осуществляется двумя разными методами:

- Снизу вверх
- Сверху вниз

Заглушка и драйвер

Инкрементальный подход осуществляется с помощью фиктивных программ, называемых заглушками и драйверами. Заглушки и драйверы не реализуют всю логику программного модуля, а только моделируют обмен данными с вызывающим модулем.

Заглушка: вызывается тестируемым модулем.

Драйвер: вызывает модуль для тестирования.

Интеграция «снизу вверх»

В восходящей стратегии каждый модуль на более низких уровнях тестируется с модулями более высоких уровней, пока не будут протестированы все модули. Требуется помощь драйверов для тестирования

Преимущества:

- Проще локализовать ошибки.
- Не тратится время на ожидание разработки всех модулей, в отличие от подхода Большого взрыва.

Недостатки:

- Критические модули (на верхнем уровне архитектуры программного обеспечения), которые контролируют поток приложения, тестируются последними и могут быть подвержены дефектам.
- Не возможно реализовать ранний прототип

Интеграция «сверху вниз»

При подходе «сверху вниз» тестирование, что логично, выполняется сверху вниз, следуя потоку управления программной системы. Используются заглушки для тестирования.

Преимущества:

- Проще локализовать баги.
- Возможность получить ранний прототип.
- Критические Модули тестируются на приоритет; основные недостатки дизайна могут быть найдены и исправлены в первую очередь.

Недостатки:

- Нужно много пней.
- Модули на более низком уровне тестируются неадекватно

Сэндвич (гибридная интеграция)

Эта стратегия представляет собой комбинацию подходов «сверху вниз» и «снизу вверх». Здесь верхнеуровневые модули тестируются с нижнеуровневыми, а нижнеуровневые модули интегрируются с верхнеуровневыми, соответственно, и тестируются. Эта стратегия использует и заглушки, и драйверы.

Как сделать интеграционное тестирование?

Алгоритм интеграционного тестирования:

1. Подготовка план интеграционных тестов
2. Разработка тестовых сценариев.
3. Выполнение тестовых сценариев и фиксирование багов.
4. Отслеживание и повторное тестирование дефектов.
5. Повторять шаги 3 и 4 до успешного завершения интеграции.

Атрибуты Интеграционного тестирования

Включает в себя следующие атрибуты:

- Методы / Подходы к тестированию (об этом говорили выше).
- Области применения и Тестирование интеграции.
- Роли и обязанности.
- Предварительные условия для Интеграционного тестирования.
- Тестовая среда.
- Планы по снижению рисков и автоматизации.

Критерии старта и окончания интеграционного тестирования

Критерии входа и выхода на этап Интеграционного тестирования, независимо от модели разработки программного обеспечения

Критерии старта:

- Модули и модульные компоненты
- Все ошибки с высоким приоритетом исправлены и закрыты
- Все модули должны быть заполнены и успешно интегрированы.
- Наличие плана Интеграционного тестирования, тестовый набор, сценарии, которые должны быть задокументированы.
- Наличие необходимой тестовой среды

Критерии окончания:

- Успешное тестирование интегрированного приложения.
- Выполненные тестовые случаи задокументированы
- Все ошибки с высоким приоритетом исправлены и закрыты
- Технические документы должны быть представлены после выпуска Примечания.

Лучшие практики / рекомендации по интеграционному тестированию

- Сначала определите интеграционную тестовую стратегию, которая не будет противоречить вашим принципам разработки, а затем подготовьте тестовые сценарии и, соответственно, протестируйте данные.
- Изучите архитектуру приложения и определите критические модули. Не забудьте проверить их на приоритет.
- Получите проекты интерфейсов от команды разработки и создайте контрольные примеры для проверки всех интерфейсов в деталях. Интерфейс к базе данных / внешнему оборудованию / программному обеспечению должен быть детально протестирован.
- После тестовых случаев именно тестовые данные играют решающую роль.
- Всегда имейте подготовленные данные перед выполнением. Не выбирайте тестовые данные во время выполнения тестовых случаев.